A Hardware Acceleration Unit for MPI Queue Processing

Keith D. Underwood, K. Scott Hemmert, Arun Rodrigues, Richard Murphy, and Ron Brightwell
Sandia National Laboratories*
P.O. Box 5800, MS-1110
Albuquerque, NM 87185-1110
{kdunder, kshemme, afrodri, rcmurph, rbbrigh}@sandia.gov

Abstract

With the heavy reliance of modern scientific applications upon the MPI Standard, it has become critical for the implementation of MPI to be as capable and as fast as possible. This has led some of the fastest modern networks to introduce the capability to offload aspects of MPI processing to an embedded processor on the network interface. With this important capability has come significant performance implications. Most notably, the time to process long queues of posted receives or unexpected messages is substantially longer on embedded processors. This paper presents an associative list matching structure to accelerate the processing of moderate length queues in MPI. Simulations are used to compare the performance of an embedded processor augmented with this capability to a baseline implementation. The proposed enhancement significantly reduces latency for moderate length queues while adding virtually no overhead for extremely short queues.

1. Introduction

In the mid-1990's, message passing became the dominant mechanism for programming massively parallel processor systems. By the late-1990's, the majority of message passing programs leveraged the MPI Standard [14]. In the intervening years, billions of dollars have been invested in developing application codes using MPI. Thus, it has become critically important to insure that new systems implement MPI as efficiently as possible.

Many approaches have been taken to characterizing the efficiency of MPI. The most common (and least useful) is to evaluate the ping-pong latency and bandwidth of the network. While these are necessary first order measures, models such as LogP [11] and LogGP [1] are more useful. Early work with these models [13] indicates that the most important thing for applications was to minimize the overhead (the time the application processor is involved in the communication). As a result, some of the highest performing networks have chosen to offload much of the work of sending and receiving MPI messages onto the network interfaces [2, 17, 16].

Unfortunately, the second largest impact on application performance is gap (the inverse of the message rate). Recent work [7, 3] has indicated that applications tend to traverse a significant number of entries in the two primary queues managed by MPI: the posted receive queue and the unexpected message queue. For networks that use embedded processors to traverse these queues, traversing long queues increases gap. Thus, a compromise has been made to decrease overhead while increasing gap in some scenarios.

This paper proposes a unique hardware structure to augment the microprocessor to accelerate list traversal and matching. The proposed hardware uses associative matching structures similar in concept to those in ternary content addressable memories (TCAMs) to perform high-performance parallel matching. These structures are enhanced with list management capabilities to support the unique combination of ordering semantics and high list entry turnover needed to support MPI point-to-point message passing.

To better understand basic properties of the design, a prototype has been created in FPGA hardware. The prototype provides an idea of both the clock rate that can be achieved and the timing that should be expected. It also serves as an avenue to explore and refine issues with the control interface. Unfortunately, this implementation would be difficult to integrate into a "real" environment. Thus, system-level simulation was used to demonstrate the usefulness of the proposed hardware. An MPI implementation was created that leverages the hardware acceleration unit. Using simulation, this MPI implementation was compared to a baseline

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

implementation using only an embedded processor with the benchmarks discussed in [22].

The following section provides further background information on the semantics of MPI and discusses other related research activities. The hardware design and software interface design are described in Sections 3 and 4, respectively. The MPI implementations and simulator are described in Section 5. The results from the comparison are presented in Section 6 and conclusions and future work are presented in Section 7.

2. Background

Conceptually, an MPI implementation has two message queues — one that contains a list of outstanding receive requests (the posted receive queue) and one that contains a list of messages that have arrived but do not match any previously posted requests (the unexpected queue). Incoming messages traverse the posted receive queue for a possible match and end up in the unexpected queue if no match is found. Before a request can be added to the posted receive queue, the unexpected queue must be searched for a possible match. This search of the unexpected queue must be an atomic operation to insure that a matching message does not arrive between the time the unexpected queue is searched and the receive is posted.

MPI messages are matched using three fields: context identifier, source rank, and message tag. The context identifier represents an MPI communicator object. This system-assigned identifier provides a safe message passing context so that messages from one context do not interfere with messages from other contexts. The source rank represents the local rank of the sending process within the communicator, and the message tag is a user-assigned value for further message selection within a context. A posted receive must explicitly match the context identifier, but may "wildcard" the source rank and message tag values to match against any value. In addition to the matching criteria, MPI also mandates an ordering constraint. Messages between two nodes in the same context must arrive in the order in which the corresponding sends were initiated.

All of the MPI implementations described in published literature represent the posted receive and unexpected queues as linear lists. Using this method, the time to traverse these queues grows linearly with the length of the list [22]. And, the length of the list can grow linearly with the number of processes in the parallel application [7, 3]. For some networks, the time spent traversing a long queue may impact the entire system, since the network interface may be unable to service other requests during the search. This can cause a poorly written or erroneous application to affect the performance of other applications in the system.

In order to reduce the search cost, approaches using hash tables have been explored [16, 21]. Hash tables can significantly reduce the time needed to find a matching entry, but can also significantly increase the time needed to insert an entry into the list. Unfortunately, this increase in insertion time has been prohibitive. The increase in insertion time for a hash relative to a list is especially noticeable in the zero-length ping-pong latency test by which high-performance networks are judged. Hashing is also complicated by the need to support wildcard matching and maintain ordering semantics. Unfortunately, an MPI implementation has no *a priori* knowledge of whether wildcard values will be used, so no application-specific approach can be taken.

The use of wildcard matching appears to be widespread. An initial analysis of several applications at Sandia has revealed that a large number use wildcards. The use of `MPI_ANY_SOURCE`, where the source of the incoming message is not known, is most prevalent. The use of `MPI_ANY_TAG` rarely occurs, perhaps since message tags are intended to be used for differentiating between specific types of messages. Re-coding applications to eliminate the use of source wildcards is non-trivial. The semantic equivalent is to post a receive from every possible source and then cancel those receives that are unused. This strategy is an inefficient use of processing and memory resources.

In this paper, we propose a hardware-based scheme for a network interface to accelerate MPI matching. Previous work has explored approaches to using network interface hardware specifically for MPI. The Quadrics QsNet [17] network has a general-purpose processor on the network interface that allows for running a user-context thread to process incoming messages. This approach allows much of the protocol processing needed to support MPI to occur on the network interface. However, the thread that implements MPI implements queues as linear lists. The network interface for the Sandia/Cray Red Storm machine [2] implements the Portals [5] programming interface, which provides protocol building blocks that support general network functionality as well as MPI efficiently. However, Portals only allows for incoming messages to traverse a linear list and there is no specific hardware to accelerate matching. There is also a significant amount of previous work on using the network interface to implement MPI collective operations efficiently [9, 8, 15]. Similarly, these approaches focus on protocol optimizations and efficient data movement operations rather than list traversal.

The hardware acceleration that we explore in this work is related to the techniques used to accelerate lookups in Internet Protocol (IP) routers. IP routers need to efficiently solve the *longest prefix match* (LPM) problem, where an incoming packet needs to be routed to the network that most closely matches its destination address. As with wildcard

values in MPI, network masks can be used to cover an entire range of addresses. For an IP router, an incoming message generates a table lookup for the closest matching destination, ultimately resulting in the selection of an outgoing port. For MPI, an incoming message causes a table lookup on the closest matching posted receive, resulting in the selection of a destination buffer for the message. A variety of software and hardware approaches have been explored to allow for quickly solving the LPM problem (see [18] for a summary), but none of them are appropriate for MPI, due mostly to the need to support the temporal ordering semantics and frequent table updates required by MPI semantics.

3. Hardware Acceleration Unit

The proposed integration of the hardware in the overall network interface chip (NIC) architecture is shown in Figure 1[1]. A typical network interface has send and receive (Tx and Rx) DMA capabilities coupled to the network through logical FIFO interfaces that provide a buffering capability. A processor with a local SRAM handles queue processing and manages transactions with the network through a local bus. The proposed new components are shown with dashed lines. To accelerate the posted receive queue, copies of the header information are provided to the associative list processing unit (ALPU) through an added FIFO. Separate command and result FIFOs are also provided to enable decoupled, asynchronous interactions with the processor. Similarly, copies of new receives being posted are fed to an ALPU that handles unexpected messages. The details of the ALPU are shown in Figure 2. The hardware is broken into three levels of hierarchy: the individual cell, a block of cells, and the overall associative matching unit.

3.1. Basic Matching Cell

At the lowest level, one of two individual cells is used in the match unit. A single posted receive queue cell is shown in Figure 2(a). The cell contains storage for both the match bits (the MPI matching information) as well as corresponding mask bits (for wildcard bits within MPI). The set of match bits can range from a pair of bits (one each for the two fields in an `MPI_Irecv` that can be wildcarded) to a full width mask as is needed by the Portals interface [5, 4, 6]. In addition, a valid bit, indicating if the entry is valid, and a tag field, used at the discretion of the software, are stored. In the implementation used here, the tag value is a 16-bit pointer to the matching entry in the local RAM.

To provide matching for the unexpected message queue, the cell is changed slightly, as shown in Figure 2(b). Rather

1 The prototype design only supports a single process, but extending it to support a limited number of processes is straightforward.

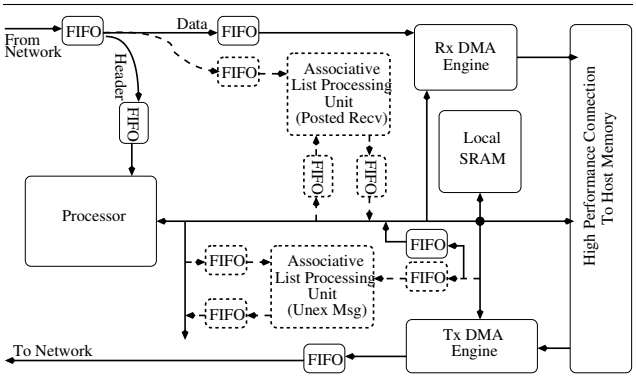


Figure 1. The proposed NIC architecture (new features shown in dashed lines)

than being stored in each cell, the mask bits are inputs. In all other respects, the cells are the same. Stored data is passed from one cell to the next. Compare logic (factoring in a set of mask bits that indicate "don't care" locations) produces a single match bit. The basic cell then has three additional outputs that feed into the higher level block. The first is a single bit that is the logical AND of the match bit and valid bit (invalid data cannot produce a valid match). The second is the tag which is muxed through priority logic to select the right match. The final output is a valid bit to allow the higher level block to manage flow control.

3.2. Block of Cells

At the next higher level (Figure 2(c)), a group of cells is combined into a cell block. In addition to a set of cells, the cell block contains a registered version of the incoming request (to facilitate timing), logic to control the flow of data, logic to correctly prioritize the tags, and logic to generate a "match location". The control flow logic drives a separate enable signal to each cell. The transfer of data from one cell to the next is enabled in two scenarios: when a match occurs and when new items are being inserted. On a successful match, MPI semantics require that the matched item be deleted; thus, the match location is broadcast to all of the cell blocks. Cells at, and below, the match location are enabled while cells above it are not, effectively deleting the matched cell and leaving the lowest priority cell empty.

During inserts, all cells are enabled if there is space available above them to compact any possible holes (which can occur during inserts but not deletes). In this implementation, "space available" means that either a higher cell in the current block or the lowest cell in the next block is empty. This is to maximize clock frequency in the FPGA prototype and is likely sufficient for all real cases. "Space avail-

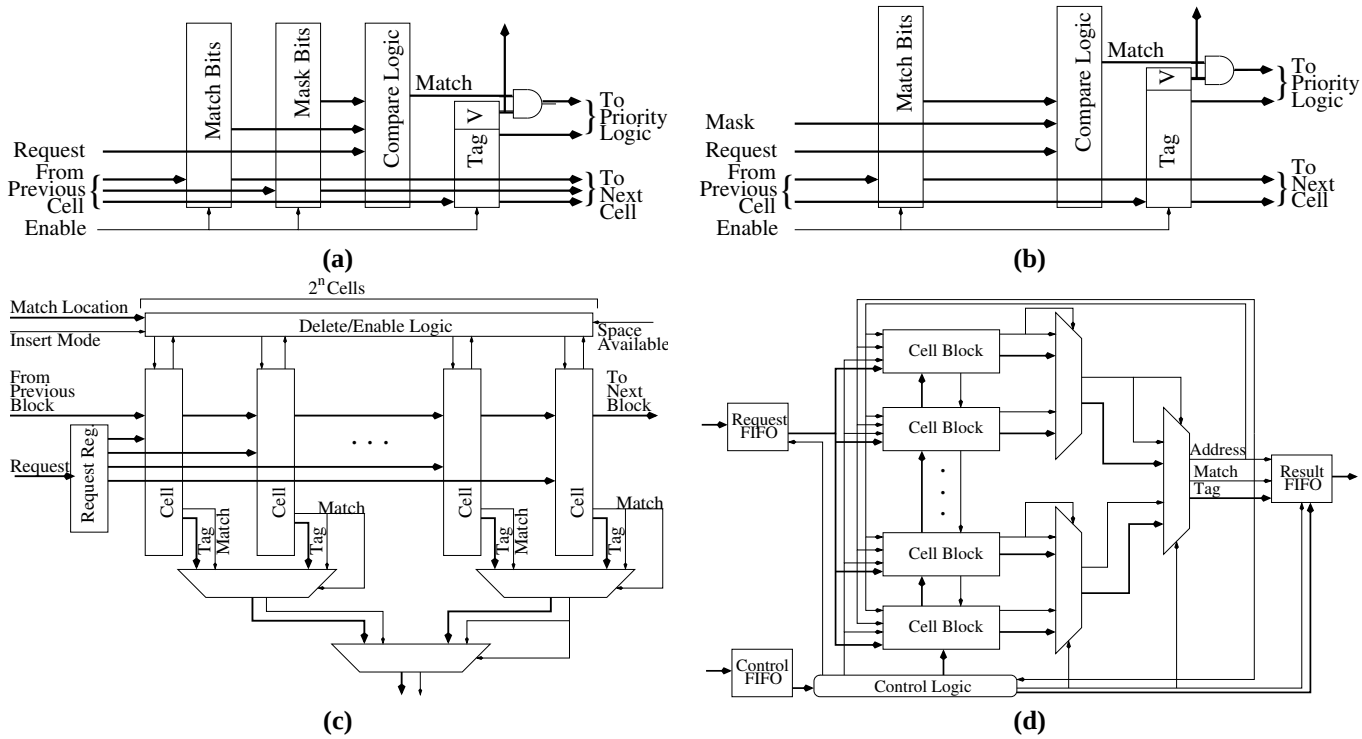


Figure 2. (a) A cell with a single match unit for the posted receive queue; (b) The corresponding cell a for the unexpected message queue; (c) A block of cells; (d) The associative match engine

able" could easily be expanded to include any cell in the next higher block or any cell in any higher block if timing constraints permitted.

The number of cells in a cell block is restricted to a power of 2 to simplify the task of prioritizing the correct tag and generating a correct match location. The prioritization logic uses the match signal to select the "correct" tag for output. In Figure 2(c), the highest order cell (furthest to the right) is the highest priority. In explanation, MPI semantics require that the first matching item in the list be considered the "correct match". In the associative matching structure, list items are inserted from the left and progress to the right. At the first level of prioritization, the higher cell in each pair of cells selects its tag if it matched and the partner tag if it did not. The match bits are also encoded as the lowest order bit of the "match location". At the second level, the logical OR of the highest order pair of match bits forms the select line for the mux and is encoded as the second lowest order bit of the match location (not shown in figure). This pattern continues through N levels of muxing for 2^N cells. The result is output as the highest order matching tag along with the encoded match location. This muxing structure could be collapsed with larger muxes, but 2-to-1 muxes improve placement regularity in the FPGA prototype.

3.3. Associative List Processing Unit

An associative list processing unit (ALPU) chains several cell blocks together and adds control logic to interface to the rest of the network interface. The cell block outputs are combined and prioritized in the same manner as cell outputs are combined in the cell block. Effectively, several cell blocks are combined to create one large, virtual array of cells. The modularization into cell blocks simplifies timing (particularly for the compaction logic) and simplifies the exploration of the design space.

The control logic in the highest level controls the interaction with the rest of the NIC. This control logic determines when new data is taken from the header input, when matches begin, when data is written to the output, when data is read from the control input, and how much space is available in the ALPU. The governing state machine is shown in Figure 3. The state machine begins in the `Match` state. In the `Match` state, the ALPU accepts a new match each time a match completes. Successful or failed matches are output to the result FIFO. If a new command arrives (the command FIFO becomes not empty), then at the completion of the current match, the state machine enters the `Read Command` state. At this point, only the `RESET` and

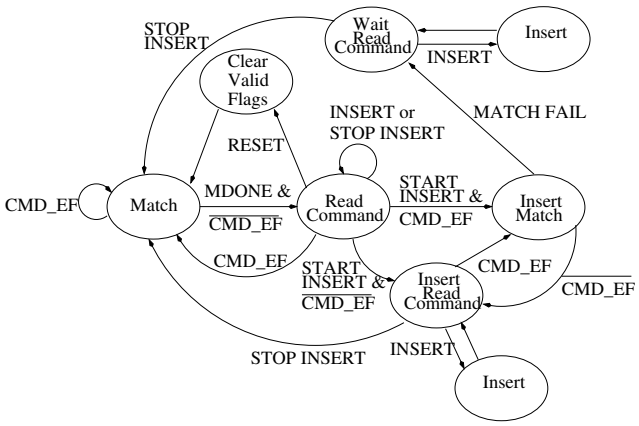


Figure 3. The controlling state machine

`START INSERT` commands are valid [2]. A `RESET` clears all of the valid flags and returns to the matching state. A `START INSERT` puts the device into insert mode. Insert mode implies a change in the matching behavior — insert commands are accepted and matching continues until a match fails. Matches are stopped temporarily for each insert (to maintain correctness), but it is likely that the processor cannot fill the command FIFO as quickly as the ALPU can drain it; thus, between inserts, matches are allowed to continue. Successful matches are output to the result FIFO and failed matches are held for a retry. This is described in further detail in Section 4. A `STOP INSERT` command returns the operation to the standard match mode.

4. Software Interface

Figure 1 provides insight into how the proposed hardware fits into the overall software architecture. In a traditional NIC[2], the header and data are separated (logically, if not physically). The processor performs list matching functions using the header information and instructs the DMA. With the proposed hardware, header data would be replicated to the associative list processing unit. The purpose of the ALPU is to quickly provide an index into the match list if a match occurs. If a match does not occur, the processor needs to decide what to do with the non-matching message (described later); thus, the processor also receives a copy of all of the header information. Components interact through FIFOs to provide hardware level decoupling that enables buffered, asynchronous operation.

2 Other commands are discarded and an empty command FIFO before a valid command causes a transition back to the match state.

4.1. Processor Interface

The ALPU requires a very limited set of commands and responses (see Table 1). A pair of commands (`START INSERT` and `STOP INSERT`) are used to instruct the ALPU to enter and exit a mode that is safe for inserts. `RESET` is used to clear the ALPU and `INSERT` is used to insert new items. Only the `INSERT` has parameters: the match bits to be used, the mask bits if needed, and a user defined tag. Certain outputs are also expected from the ALPU. The `START ACKNOWLEDGE` is returned in response to a `START INSERT` command and indicates the number of free slots in the ALPU. `MATCH SUCCESS` and `MATCH FAILURE` are the responses that are expected in normal ALPU operation. General operation of the device proceeds as follows. A `START INSERT` and its response (`START ACKNOWLEDGE`) must occur before an `INSERT` can be performed. Inserts may then be performed until a `STOP INSERT`. `MATCH SUCCESS` can occur at any time, but `MATCH FAILURE` cannot occur between a `START ACKNOWLEDGE` and a `STOP INSERT`.

4.2. Overall List Management

To manage MPI queues using an ALPU, a microprocessor must apply appropriate heuristics. As previously shown [7, 3], these queues can grow to hundreds of items; however, at times, the queues can also be quite short. Because using the ALPU will incur a certain amount of overhead, the software must only use it when the queue is adequately long. In addition, the software must recognize that inserting elements into the ALPU incurs overhead and should attempt to conglomerate insertions into the unit.

Even though the ALPU will provide high-performance matching, the processor should keep a copy of the list. The list copy allows the ALPU to return a pointer to a list entry, instead of the entire entry. As entries are matched (and, thus, deleted in the ALPU), the processor's copy of the entry must also be deleted. Furthermore, the processor may have entries that have not yet been entered into the ALPU. A pointer to the start of the portion of the list that has not been entered into the ALPU should also be maintained for proper handling of responses (discussed further in Section 4.4).

4.3. Match Entry Insertion

When the hardware is initialized, the ALPU is empty. Matching is enabled, but no matches will succeed; thus, the hardware should be designed such that the processor can disable the delivery of duplicate information (headers or new posted receives) to the ALPU until it is initialized. As new entries for the queue arrive (new posted receives or unexpected messages), the processor should build the ap-

Command	Description	Inputs
START INSERT	Instruct the ALPU to enter insert mode	None
INSERT	Insert a new entry in the ALPU	Match bits, Mask bits (optional), and tag
STOP INSERT	Instruct the ALPU to exit insert mode	None
RESET	Clear all entries in the ALPU	None

Response	Description	Outputs
START ACKNOWLEDGE	ALPU has entered insert mode	Number of free entries
MATCH SUCCESS	Input matched list item	Tag from list item matched
MATCH FAILURE	Input did not match list item	None

Table 1. Associative list processing unit commands and responses

propriate queue in memory. When the queue length crosses a threshold (defined empirically to enable optimal performance), the processor sends a `START INSERT` command to the ALPU. To avoid a potential race condition where a match in the pipeline fails while the processor is performing an insert (inserts are irrevocable), the processor must wait for an `INSERT ACKNOWLEDGE` response. In response to the `START INSERT` command, the ALPU enters a safe state where matches can occur, but matches that fail are held for retry until after all inserts complete.

While waiting for the `INSERT ACKNOWLEDGE`, the processor may receive `MATCH SUCCESS` or `MATCH FAILED` responses, which must be handled as described in the next section. The `INSERT ACKNOWLEDGE` will include the number of entries it is safe to insert. The processor should also track this number to insure that it does not attempt to start inserting when little or no space is available. Having received the `INSERT ACKNOWLEDGE`, the processor should insert all items as quickly as possible and then send a `STOP INSERT` command. If the number of items to be inserted is large, the processor may need to periodically clear the result FIFO of successful matches that occur during the insert process to prevent it from filling. Each insert includes the information to be matched, optionally a set of mask bits (for posted receives), and a tag. The tag can be any value and will be returned on a successful match; however, the recommended use is to store a pointer to the corresponding queue entry in local RAM.

4.4. Result Handling

If the ALPU is enabled, the processor must retrieve a response from the ALPU for every header that is received. The processor should first retrieve the copy of the data provided to it and then retrieve the response. The response will either be a `MATCH SUCCESS` or a `MATCH FAILED`. On a success, the returned tag can be used to point directly to the matching list item in the processor's copy of the list. On a failure, the processor must use the local copy of the data and search the portion of the list that is not in the ALPU. If there is still no match, the data must be handled correctly. If the data is a header that did not match an item in the posted receive queue, it should be inserted into the unexpected message queue. If the data is a new posted receive that did not match an item in the unexpected message queue, it needs to be added to the posted receive queue.

5. Methodology

This research began with a benchmark that exposed a limitation of modern network interfaces cards (NICs) that leverage embedded processors. The behavior was replicated using a simulation of a modern system environment (including NIC). The simulated NIC was enhanced with the proposed associative list processing unit (ALPU) and the MPI implementation was modified to leverage the feature. Finally, an independent hardware prototype was created to understand the performance of the proposed design.

5.1. Benchmarks

The primary motivation for this design was to reduce message latency when long posted receive queues or long unexpected message queues were present. The magnitude of the problem was revealed in an earlier study [22] using two newly designed benchmarks. These benchmarks are used again here to study the impacts of the ALPU. The benchmark that measures the impact of the pre-posted receive queue length provides three degrees of freedom: the length of the pre-posted receive queue, the portion of the pre-posted receive queue that is traversed, and the size of the message. This enables the user to measure the impacts of both the receive queue length and the actual queue traversal. The benchmark that assesses the impact of unexpected message queue length on message latency only allows the length of the unexpected message queue and the size of the message to be varied. It deviates from the traditional way of

Parameter	CPU	NIC Processor
Fetch Q	4	2
Issue Width	8	4
Commit Width	4	4
RUU Size	64	16
Integer Units	4	2
Memory Ports	3	1
L1 Caches	64K 2-way	32K 64-way
L2 Cache	512K	none
Clock Speed	2Ghz	500Mhz
Lat. To Main Memory	80-85ns	115-120ns
ISA	PowerPC	PowerPC
Network Wire Lat.	200 ns	

Table 2. Processor Simulation Parameters

measuring latency in that it includes the time to post the receive for the latency measuring message as part of the latency. This better reflects the way that MPI is actually used by applications, which typically have some number of iterations and post receives in each iteration.

5.2. Simulation Environment

System-level simulation of the matching structure used a simulator based on Enkidu [19], a component-based discrete event simulation framework. To simulate the CPU and NIC processors, `sim-outorder` from the SimpleScalar [10] tool suite was integrated into this framework. Components representing a simple network, DMA engines, a memory controller, and DRAM chips were added. The memory hierarchy was modeled to include contention for open rows on the DRAM chips.

The main processor was parameterized to be similar to a modern high-performance processor. The NIC processor was parameterized to be similar to a processor in higher end network cards, such as the PowerPC 440 (see table 2). A simple bus on the NIC connected the main processor with the DMA engine, SRAM, and matching structure. This bus was simulated with a 20ns delay. The SRAM was modeled with a 3ns delay.

5.3. MPI Implementation

The prototype MPI implements a subset of MPI-1.2 [14]. With the exception of `MPI_Barrier()`, only basic point-to-point communication and basic support functions were implemented. Only support for basic MPI Datatypes is included and `MPI_COMM_WORLD`, is the only group. The MPI was implemented in roughly 1600 lines of C++ and compiled with GNU g++ 3.3.

The primary data structures are a series of linked lists to contain requests and the state required to advance them. All of the data structures reside in the NIC memory, and the NIC handles most of the MPI processing including the management of the unexpected message queue and the posted receive queue. Further details of the MPI implementation can be found in [20].

Use of the ALPU requires minimal modification of the basic MPI implementation. Each iteration of the NIC's main loop updates the posted receive ALPU and the unexpected ALPU. A pointer is kept to indicate which portions of the `postedRecvQ` and `unexpectedQ` have been transfered to the ALPU and which have not. If there are portions of these lists that have not yet been added to the ALPU, the NIC will attempt to insert them. The NIC sends a `START INSERT` message, and then drains the ALPU's result FIFO of any match results until a `START ACKNOWLEDGE` is received. It then attempts to insert as many of the remaining headers as it can, updating the pointer to indicate which portions of the queue have been inserted. After the inserts, it will send a `STOP INSERT` command.

When an incoming message arrives, its header is automatically sent to the ALPU. When the NIC detect this message, it checks the ALPU's output queue to see if it has matched. If it does, the relevant request is removed from the `postedRecvQ`. If no match is found, the portion of the `postedRecvQ` that is not on the ALPU is checked. If no match is found, the message header is added to the `unexpectedQ` and will be inserted into the unexpected message queue ALPU. Similarly, when receive requests arrive, the unexpected ALPU is checked to see if a match has occurred. If the ALPU returns `MATCH FAILURE`, any portion of the `unexpectedQ` not on the ALPU is checked.

5.4. FPGA Prototype

To provide a reasonable estimate of the size and operating frequency of the ALPU, a prototype implementation was created, targeting Xilinx Virtex 2 and Virtex 2 Pro FPGAs. The ALPU was designed using JHDL [12], a structural design tool that provides fine-grained control over the placement of logic on the FPGA. The final design is parameterized to allow different match and tag widths, as well as different combinations of the total number of cells and the number of cells in each block.

When designing the unit, the top priorities were small area, high speed and regularity in placement. To allow for higher operating frequencies, the ALPU requires multiple clock cycles to complete a match (6 or 7 depending on the size of the ALPU and the blocking factor). If desired, it is possible to overlap execution of the first and last cycles. The simulation results assume a 7 cycle pipelining latency with

Total Cells	Block Size	Size LUTs	FFs	Slices	Speed (MHz)
256	8	17,372	28,908	15,766	112.5
	16	17,573	27,656	15,090	111.4
	32	18,054	26,971	14,742	100.2
128	8	8,687	14,562	7,945	111.5
	16	8,786	13,897	7,606	112.1
	32	9.025	13,605	7,431	100.6

Table 3. Posted Receive ALPU performance

Total Cells	Block Size	Size LUTs	FFs	Slices	Speed (MHz)
256	8	17,339	19,414	11,562	112.1
	16	17,556	17,490	10,631	111.9
	32	18,045	16,469	10,350	100.9
128	8	8,672	9,773	5,806	111.2
	16	8,777	8,771	5,356	112.1
	32	9,020	8,311	5,215	100.6

Table 4. Unexpected Msg. ALPU performance

no overlap of execution. The current design also allows inserts to happen on every other clock cycle.

6. Results

Three sets of experiments were performed. The first was an FPGA-based prototype used to explore size and performance issues of the design. The second experiment simulated the performance of a NIC with and without the associative list processing unit (ALPU) for the posted receive queue. In the final experiment, the ALPU was applied to the management of the unexpected message queue. Results from these experiments indicate that the ALPU is small and fast enough, and provides sufficient benefits to be practical.

6.1. FPGA Prototype

Prototypes for list units accelerating both posted receives and unexpected messages were created. The Xilinx FPGA tool chain was used to map the prototypes to a Virtex-II Pro 100 FPGA with a -5 speed grade[3]. We chose to test units with both 256 and 128 total cells, with block sizes of 8, 16, and 32. For each test, the match width was set to 42 and the tag width was 16. These widths are adequate to support an MPI implementation supporting the full specification on a 32K node system. In addition, there is a mask bit for every match bit[4]. The sizes and speeds of the prototypes are found in Tables 3 and 4. The size and speed numbers were provided by the Xilinx tools. The sizes include the number of 4-input lookup tables (LUTs), the number of flip flops (FFs), as well as the number of slices[5].

Though the largest ALPU consumes 35% of a large FPGA, as an ASIC, the size would be similar to that of commercially available ternary CAMs. In addition, The move to standard cell ASIC technology would provide a $5\times$ increase in clock frequency (an extremely conservative estimate). This means that the prototypes would all run at 500MHz — the core logic speed in the ASC Red Storm network interface[2]. The implied size and speed of the ALPU in an ASIC makes it a good candidate for addition to a network interface offload engine.

3 This is the slowest speed grade on a 0.13 micron process.

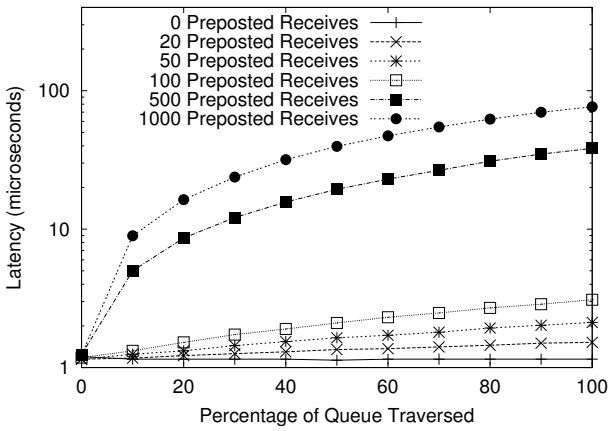
4 Providing a mask bit for each match bit increases configurability and supports protocols beyond MPI, such as Portals.

5 A slice has two LUTs and two FFs, but often are not used this densely.

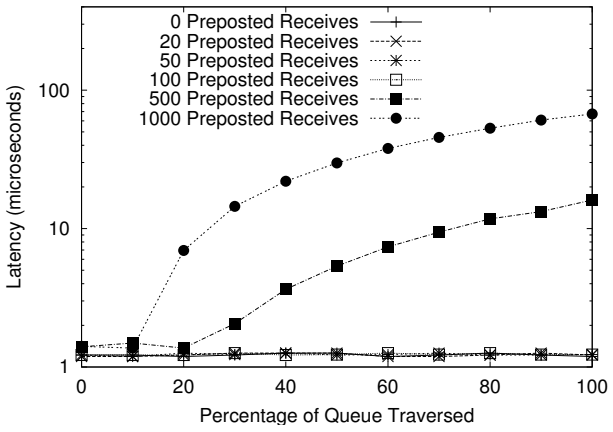
6.2. Preposted Latency Impacts

Figure 4 compares the performance of a baseline NIC to the same NIC enhanced with a 128-entry ALPU and a 256-entry ALPU. On the left, the full 3D surface is shown for each configuration while the right shows projections of several of the lines on a 2D graph. The graphs have some interesting traits. For the baseline NIC (parts (a) and (b)), the low end of the graph shows each entry traversed adding an average of 19 ns of latency. By comparison, for a Quadrics Elan4 NIC, each entry traversed adds 150 ns of latency. The almost $10\times$ performance improvement is not surprising because the NIC being modeled has a significantly faster clock ($2.5\times$), is dual issue (for integers, floating-point does not get used), and has separate 32 KB instruction and data caches. When the queue is too long to fit in cache, the average time per entry traversed grows to 75 ns. This overhead shows up even when the entire list is not traversed. For example, the time to traverse an entire 400-entry list is $18\mu s$ and the time to traverse 80% of a 500-entry list is $30\mu s$.

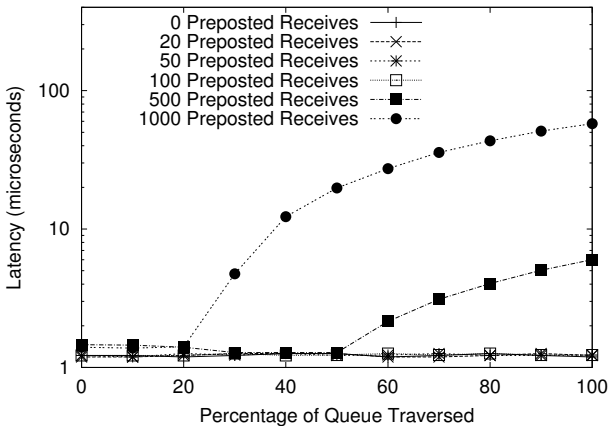
Incorporating an ALPU yields two significant advantages. The most dramatic advantage is a flat latency curve until the length of the queue traversal crosses the size of the ALPU. The penalty is a 40 ns increase in the baseline latency (zero-length posted receive queues) as the processor incurs overhead from interacting with the ALPU. With 4 entries in the posted receive queue, the ALPU breaks even. Thus, an MPI library could be optimized to not use the ALPU until the list is at least 4 entries long. The second advantage provided by the ALPU is the reduction in the usage of the cache. By using the ALPU, the processor is not required to traverse the first N entries of the queue, even if the ALPU does not find a match. The storage required by the ALPU is relatively small (the entire queue entry does not have to be stored). Each entry in the ALPU contains match-



(a)



(b)



(c)

Figure 4. (a) Growth of latency with standard posted receive queue; (b) Growth of latency using a 128-entry ALPU to manage posted receive queue; (c) Growth of latency using a 256-entry ALPU to manage posted receive queue

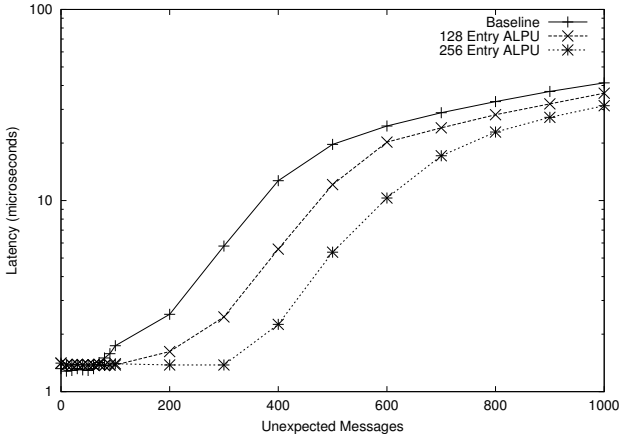


Figure 5. Growth of latency with unexpected queue length

ing data only, but the processor stores several other pieces of data in the queue entry. Thus, the number of cache lines the processor must retrieve from memory is dramatically reduced if it does not have to search the first several entries.

6.3. Unexpected Message Impacts

In contrast to the preposted queue, the results show no advantage from the ALPU for applications that have extremely short unexpected message queues (Figure 5). Indeed, with short unexpected message queues, the ALPU appears to show a small increase in latency (tens of nanoseconds). After the unexpected queue reaches a length of 70 entries, the ALPU begins to offer a significant advantage. As with the preposted queue, as the NIC processor cache is exhausted, latency rises more dramatically. The ALPU is still able to delay this rapid rise. These graphs do not tell the full story of the ALPU advantage. To be conservative, the benchmark is written to allow the overlap of posting a receive with message transfer. In real life, a long posted receive queue is created by pre-posting several receives consecutively (without matches arriving). Each receive would take progressively longer and would impact the application execution time directly. In such a case, the ALPU would offer a much greater benefit.

7. Conclusions and Future Work

Both the posted receive queue and the unexpected message queue can be significant bottlenecks in the processing of MPI messages. This paper presents a novel feature to be integrated in a network interface to accelerate the pro-

cessing of both of these critical queues in MPI. The associative list processing unit (ALPU) was prototyped in an FPGA and was found to be small enough and fast enough to be integrated in a modern network interface.

To assess the performance impact of the ALPU, simulations of both a baseline NIC and a NIC enhanced with the ALPU were performed. The addition of the ALPU added minimal overhead, even when used on extremely short queues. As the queue length grew, the use of the ALPU achieved dramatic drops in message latency. Even when the queue length grows beyond the size of the ALPU, the addition of the ALPU is an inexpensive way to decrease the pressure on the cache for the processor in the NIC.

The optimization of MPI is a broad and ongoing effort. Focus areas include other optimization techniques to further accelerate queue traversal and techniques to traverse queues quickly with fewer hardware resources. Another area of research will focus on how to offload significant portions of the Portals interface to enable support of MPI, run-time software, and I/O.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Sheiman. LogGP: Incorporating long messages into the LogP model. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[2] R. Alverson. Red Storm. In *Invited Talk, Hot Chips 15*, August 2003.

[3] R. Brightwell, S. Goudy, and K. D. Underwood. A preliminary analysis of the MPI queue characteritics of several applications. *submitted*, May 2004.

[4] R. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[5] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[6] R. Brightwell, A. B. Maccabe, and R. Riesen. Design, implementation, and performance of MPI on Portals 3.0. *International Journal of High Performance Computing Applications*, 17(1):7–20, Spring 2003.

[7] R. Brightwell and K. D. Underwood. An analysis of NIC resource usage for offloading MPI. In *Proceedings of the 2004 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April 2004.

[8] D. Buntinas and D. K. Panda. NIC-based reduction in Myrinet clusters: Is it beneficial? In *Proceedings of the SAN-02 Workshop (in conjunction with HPCA)*, February 2002.

[9] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[10] D. Burger and T. Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[11] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[12] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–24, Napa, CA, April 1999. IEEE Computer Society, IEEE.

[13] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[14] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[15] A. Moody, J. Fernandez, F. Petrini, and D. K. Panda. Scalable NIC-based reduction on large-scale clusters. In *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[16] Myricom, Inc. Myrinet Express (MX): A high performance, low-level, message-passing interface for Myrinet, July 2003.

[17] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/February 2002.

[18] B. Plattner, G. Varghese, J. Turner, and M. Waldvogel. Scalable high-speed prefix matching, February 2002.

[19] A. Rodrigues. Enkidu discrete event simulation framework. Technical Report TR04-14, University of Notre Dame, 2004.

[20] A. Rodrigues, R. Murphy, R. Brightwell, and K. Underwood. Enhancing NIC performance for MPI using processing-in-memory. In *Proceedings the 2005 Workshop on Communication Architectures for Clusters*, April 2005.

[21] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing. In *Proceedings of the 2001 Conference on Supercomputing*, Nov. 2001.

[22] K. D. Underwood and R. Brightwell. The impact of MPI queue usage on message latency. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.